arXiv:1402.5687v1 [cs.LO] 23 Feb 2014

Monoidal computer II: Normal complexity by string diagrams

(Extended abstract)

Dusko Pavlovic
University of Hawaii
Email: dusko@hawaii.edu

Abstract

In Monoidal Computer I, *we introduced a categorical model of computation where the formal reasoning about computability was supported by the simple and popular diagrammatic language of* string diagrams. *In the present paper, we refine and extend that model of computation to support a formal complexity theory as well. This formalization brings to the foreground the concept of* normal *complexity measures, which allow decompositions akin to Kleene's normal form. Such measures turn out to be just those where evaluating the complexity of a program does not require substantially more resources than evaluating the program itself. The usual time and space complexity are thus normal measures, whereas the average and the randomized complexity measures are not. While the measures that are not normal provide important design time information about algorithms, and for theoretical analyses, normal measures can also be used at run time, as practical tools of computation, e.g. to set the bounds for hypothesis testing, inductive inference and algorithmic learning.*

Index Terms

categorical models and logics, foundations of computability, logical aspects of computational complexity, semantics of computation

1. Introduction

Motivation. This is an old fashioned paper, inspired by Blum's *"Machine Independent Complexity Theory"* [6], Levin's and Meyer's work on the *"Fundamental Theorem of Complexity"* [23], [25], and a host of such papers — mostly from the 1970s. Why is it worthwhile to go back to that work, when theoretical computer science went in a different direction?

Why is it reasonable that theoretical computer science still works with tapes, and still programs its abstract machines in low level machine languages, long after the real computers stopped using tapes, and when the real programmers only use machine languages to program firmware? There are surely some reasonable and convincing answers to this question, and there are probably some good reasons why the theoreticians like to use low level models. But as theoretical computer science is becoming more and more practical, the practical tasks are emerging where high level models and machine independent reasoning and programming are becoming necessary. One such task [29] led me on the path towards monoidal computer. The task was to measure the hardness of deriving an attack algorithm on a given system with known vulnerabilities. This task is easily formalized, but requires measuring the complexity of algorithm transformations. There are high level programming languages convenient for programming program transformations, but there are no research tools for studying the complexity of such programs. Wondering why, I turned to a tool that seemed useful for understanding high level programming languages: *category theory*.

So my defense for the strange and demanding concoction of the formalisms that I offer here is very ambitious: I am hoping that it will get us a step closer to a high level language for reasoning about computability, complexity and cryptography, by reusing some ideas and structures that evolved in such languages for reasoning about software, systems, and even about quantum computing. If this turns out to be a completely wrong direction, then we shall at least gain some insight why the problem of complexity is so different from pretty much everything else.

Idea. Intuitively, computation is often viewed as a straightforward process. A computer is given a program F, and it is set into a configuration $\langle q, a\rangle$,

where q is the initial state, and a are the input data. The computer then searches for an instruction of the program F that is enabled by the configuration $\langle q, a\rangle$, and if it finds such an instruction, it executes it, thereby changing the state to q_1 and the data to a_1. Then it searches for an instruction enabled by the configuration $\langle q_1, a_1\rangle$, which leads to a configuration $\langle q_2, a_2\rangle$, and so on. The computer thus builds an *execution trace*, which can be viewed as an expression in the form

$$F : \langle q, a\rangle \to \langle q_1, a_1\rangle \to \cdots \to \langle q_i, a_i\rangle \to \cdots \quad (1)$$

If the trace reaches a configuration $\langle q_n, a_n\rangle$ where no instructions of the program F are enabled, then the computation terminates. The output of the computation can then be found among the data a_n, say as the part that is stored on a designated output tape.

Kleene's Normal Form Theorem [19] formalizes this view of computation mathematically. It says that every computable function $f : A \to B$, implemented by some program F, can be reduced to the *normal form*

$$f(a) \quad = \quad w\big(\mu x.\ T(F, a, x)\big) \quad (2)$$

where $w : \mathbb{N} \to \mathbb{N}$ and $T : \mathbb{N}^3 \to \{0, 1\}$ are some primitive recursive functions, explained below, and $\mu x : \{0, 1\}^{\mathbb{N}} \to \mathbb{N}$ is the search operator. What does (2) mean? The idea is that any execution trace (1), as soon as it is finite, and thus denotes a terminating computation, can be encoded by a unique natural number x. Kleene constructed a primitive recursive predicate $T(F, a, x)$ that tests whether x encodes the trace (1) of an execution of the program F on data a. If the search $\mu x.T(F, a, x)$ finds such a trace x, then the primitive recursive function w extracts from its final configuration the output of the computation of F on a. Formula (2) thus tells that every computation can be reduced to a single search μx, precomposed with a primitive recursive predicate $T(F, a, x)$ which tests that x is the trace of the program F on the input a, and postcomposed with a primitive recursive function w which extracts the outputs from the traces.

Note, however, that besides the program, the input and the output, the trace (1) also carries the information how many steps did the computation take, and what was the largest memory area that it occupied. So if we replace the primitive recursive function w with some other functions, we can compute the time and the space complexity. That is the idea that we pursue in this paper. It arises from the observation that the suitable encodings of execution traces are not only complexity measures themselves, but that they are universal among a natural family of complexity measures, which we call normal. We present them as a categorical structure, and calculate with them using string diagrams.

Outline of the paper. In Sec. 2, we spell out the minimal preliminaries that fit in this paper. In Sec. 3, we review the structure of monoidal computer. Sec. 4 introduces the structure of *graded monoidal computer*. The grades implement the execution traces categorically. In Sec. 5, we spell out the normalization in graded monoidal computer. In Sec. 6 we define and characterize normal complexity measures.

Related work. While computability and complexity theorists seldom felt a need to learn about categories, there is a rich tradition of categorical research in computability theory, starting from one of the founders of category theory and his students [12], [26], through the extensive categorical investigations of realizability [16], [5], [15], to the recent work on Turing categories [9], and on a monoidal structure of Turing machines [3]. This recent work has, of course, interesting correlations with basic monoidal computer, but also substantial differences, arising from the different goals. The closest in spirit to the present work seems [2], also drawing its structural content from abstract complexity theory.

2. Preliminaries

A monoidal computer will be a *symmetric monoidal category*, with some additional structure. As a matter of convenience, and with no loss of generality, we assume that it is a *strict* monoidal category. The reader familiar with these concepts may wish to skip to the next section. For the casual reader unfamiliar with these concepts, we attempt to provide enough intuitions to understand the presented ideas. The reader interested to learn more about monoidal categories should consult one of many textbooks, e.g. [24], [18].

Monoidal categories. Intuitively, a monoidal category is a category $\mathcal{C}$ together with a functorial monoid structure

$$\mathcal{C} \times \mathcal{C} \xrightarrow{\otimes} \mathcal{C} \xleftarrow{I} \mathbb{1}$$

When $\mathcal{C}$ is a monoidal *computer*, then we think of its objects $A, B, \ldots \in |\mathcal{C}|$ as datatypes, and of its morphisms, $f, g \ldots \in \mathcal{C}(A, B)$ as computations. The tensor product $A \otimes P \xrightarrow{f\otimes t} B \otimes Q$ then captures the parallel composition of the computations $A \xrightarrow{f} B$ and $P \xrightarrow{t} Q$, whereas the categorical composition $A \xrightarrow{f;g} C$ is the sequential composition of $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$.

With no loss of generality, we assume that the tensors are *strictly* associative and unitary, and thus

treat the objects $A \otimes (B \otimes C)$ and $(A \otimes B) \otimes C$ as the same, and do not distinguish $A \otimes I$ and $I \otimes A$ from A. This allows us to elide many parentheses and natural coherences [17], [24, Sec. VII.2]. Note, however, that the isomorphisms $A \otimes B \xrightarrow{S} B \otimes A$ cannot be eliminated without causing a degeneracy.

Notations. When no confusion seems likely, we write

- AB instead of $A \otimes B$
- $\mathcal{C}(X)$ instead of $\mathcal{C}(I, X)$

String diagrams. A salient feature of monoidal categories is that the algebraic laws of the monoidal structure correspond precisely and conveniently to the geometric laws of *string diagrams*, formalized in [17], but going back to [34]. See also [38] for survey. A string diagram usually consists of polygons or circles linked by strings. In a monoidal computer, the polygons represent computations, whereas the strings represent data types, or the channels through which the data of the corresponding types flow. String diagrams thus display the data flows throught composite computations. The reason why string diagrams are convenient for this is that the two program operations that usually generate data flows, the sequential composition $f\,;g$ and the parallel composition $f \otimes t$, precisely correspond to the two geometric operations that generate string diagrams: one is the operation of connecting the polygons $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$ by the string B, whereas the other one puts the polygons $A \xrightarrow{f} B$ and $P \xrightarrow{t} Q$ next to each other without connecting them. The associativity of these geometric operations then imposes the associativity law on the corresponding operations on computations. The identity morphism id_A, as the unit of the sequential composition, can be viewed as the channel of type A, and can thus be presented as the string A itself, or as an "invisible polygon" freely moved along the string A. The unit type I can be similarly presented as an "invisible string", freely added and removed to string diagrams. The algebraic laws of the monoidal structure are thus captured by the geometric properties of the string diagrams. The string crossings correspond to the symmetries $A \otimes B \xrightarrow{S} B \otimes A$.

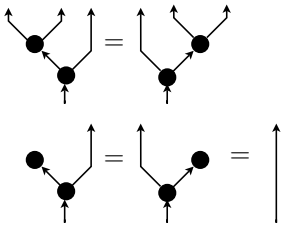
Data services. We call *data service* the monoidal structure that allows passing the data around. In computer programs and in mathematical formulas, the data are usually passed around using variables. They allow copying and propagating the data values where they are needed, or deleting them when they are not needed. The basic features of a variable are thus that it can be freely copied or deleted. The basic data services over a type A in a monoidal category $\mathcal{C}$ are

- the *copying* operation $A \xrightarrow{\delta} A \otimes A$, and
- the *deleting* operation $A \xrightarrow{\top} I$,

which together form a *comonoid*, i.e. satisfy the equations

$$\delta\,;(\delta \otimes A) = \delta\,;(A \otimes \delta) \tag{3}$$

$$\delta\,;(\top \otimes A) = \delta\,;(A \otimes \top) = \mathrm{id}_A \tag{4}$$



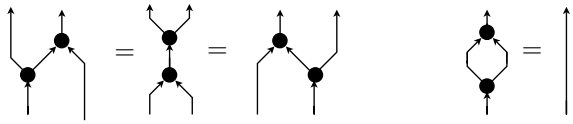
The correspondence between the variables and the comonoids was formalized and explained in [27], [31]. The associativity and the unit of the copying operation allow defining the unique n-ary copying operations $A \xrightarrow{\delta} A^{\otimes n}$, for all $n \geq 0$. The tensor products $\otimes$ in $\mathcal{C}$ are the cartesian products $\times$ if and only if every A in $\mathcal{C}$ carries a canonical comonoid $A \times A \xleftarrow{\delta} A \xrightarrow{\top} \mathbb{1}$, where $\mathbb{1}$ is the final object of $\mathcal{C}$, and all morphisms of $\mathcal{C}$ are comonoid homomorphisms, or equivalently, the families $A \xrightarrow{\delta} A \times A$ and $A \xrightarrow{\top} \mathbb{1}$ are natural. Cartesian categories are thus just the categories with natural copying and deleting operations.

But besides copying and deleting, we often also need a third data service:

- the *comparison* operation $A \otimes A \xrightarrow{\varrho} A$

which is required to be associative, in the sense dual to (3), and thus makes A into a *semigroup*. Its associativity allows defining the unique n-ary comparisons $A^{\otimes n} \xrightarrow{\varrho} A$, for $n \geq 1$. The copying and the comparison operations are further required to satisfy the *data distribution* (or *Frobenius* [8]) conditions

$$(\delta \otimes A)\,;(A \otimes \varrho) = \varrho\,;\delta = (A \otimes \delta)\,;(\varrho \otimes A) \qquad \delta\,;\varrho = \mathrm{id}$$



These conditions allow factoring any morphism $A^{\otimes m} \to A^{\otimes n}$ generated by δ and ϱ into the *spider* form $A^{\otimes m} \xrightarrow{\varrho} A \xrightarrow{\delta} A^{\otimes n}$. In summary,

Definition 2.1: A *data service* over an object A of a monoidal category $\mathcal{C}$ consists of

- the *copying* operation $A \xrightarrow{\delta} A \otimes A$,
- the *deleting* operation $A \xrightarrow{\top} I$, and
- the *comparison* operation $A \otimes A \xrightarrow{\varrho} A$

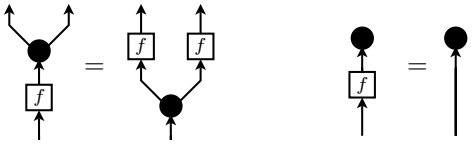
such that

- δ and ϱ are associative,
- $\top$ is the unit of δ,
- δ and ϱ satisfy the data distribution conditions.

Examples and non-examples of data services. Non-trivial cartesian categories do not support comparisons. However, the cartesian structures $A \times A \xleftarrow{\delta} A \xrightarrow{\top} \mathbb{1}$ from the category Set of sets and functions also live in the monoidal categories Rel of sets and relations and Pfn of sets and partial functions. These categories are not cartesian because the singleton maps are only natural with respect to the total relations, whereas the diagonals are only natural with respect to the single-valued relations (i.e. partial functions). Both Rel and Pfn allow comparisons: a comparison $A \times A \xrightarrow{\varrho} A$ is the converse of the diagonal, i.e. $\varrho(x,x) = x$, and $\varrho(x,y)$ remains undefined if $x \neq y$. So the cartesian structure of Set provides the standard data services in Rel and Pfn. In addition, Rel also admits many nonstandard data services, that do not come from the cartesian structure. This was analyzed in [28], [31]. Indeed, a Any abelian group (or groupoid) structure $A \times A \xrightarrow{+} A$ can be used as the comparison operation, with the corresponding copying operation $A \xrightarrow{\delta} A \times A$ relating each $x \in A$ with all pairs $\langle y, z\rangle \in A \times A$ such that $x = y + z$. The deletion operation $A \xrightarrow{\top} \mathbb{1}$ relates the unit of the group A with the only element of $\mathbb{1}$.

Functions. A morphism $f \in \mathcal{C}(A,B)$ is called *function* if it is a comonoid homomorphism with respect to the data services on A and B, i.e. if it satisfies the following equations

$$f;\delta_B \;=\; \delta_A;(f \otimes f) \qquad f;\top_B \;=\; \top_A \quad (5)$$



If $\mathcal{C}$ is the category of relations, then the first equation says that f is a single-valued relation, whereas the second equation says that it is total. Hence the name.

Notation. Assuming that $\mathcal{C}$ is given with a chosen data service on every objects, we denote by $\mathcal{C}^\natural$ the category of functions in $\mathcal{C}$.

Note that the morphisms δ and $\top$ from the data services are functions with respect to the induced data services. They are thus contained in $\mathcal{C}^\natural$, and they are natural with respect to the functions. It follows that $\mathcal{C}^\natural$ is a cartesian category.

3. Monoidal computer

3.1. Background and definition

From [21] to [20], the structure of *cartesian closed categories* has been the foundation of categorical logic and type theory. It is based on the correspondence:

$$\mathcal{C}(X, B^A) \quad \cong \quad \mathcal{C}(X \times A, B) \qquad (6)$$

between the functions $X \times A \xrightarrow{f(x,a)} B$ on the right hand side, and their abstractions $X \xrightarrow{\lambda a.f(x,a)} B^A$ on the left. In a practice-oriented semantics of computation, the λ-abstraction could be used to represent programming, as an operation mapping the specifications of the computable functions on the right to the programs on the left. The program evaluation, as an operation mapping the programs to computable functions, would then be represented by the application, i.e. the transition from left to right in (6). But since the correspondence in (6) is bijective, these transitions don't just evaluate each program into a unique computable function, but also assign a unique program to each computable function. In reality, of course, there are always infinitely many different programs that compute the same function, as soon as the programming language can express enough arithmetic. The *extensional* models of computation, underlying (6), can be interpreted as viewing the computer as a black box, where only the inputs and the outputs are observable, and any two programs that map the same inputs to the same outputs are indistinguishable. Such view has been not only the tenet of the theory of *denotational* semantics [37], [39], but also the stepping stone into the practice of functional programming [40].

The *intensional* view of computation can also be presented categorically in many ways [12], [26], [9], as mentioned in the Introduction. One obvious way to allow multiple programs for the same computable function is to relax the bijection in (6) to a surjection. This surjection is the main structural component of monoidal computer. After some fine tuning, it takes the form

$$\mathcal{C}^\natural(X, \mathbb{P}) \quad \xrightarrow{\gamma_X^{AB}} \!\!\!\!\rightarrow \quad \mathcal{C}(X \otimes A, B) \qquad (7)$$

where the program enumerations $X \xrightarrow{F_x} \mathbb{P}$ on the left are mapped to the computations $X \otimes A \xrightarrow{\{F_x\}(a)} B$ on the right. Here the Kleene bracket $\{-\}$ executes the program F_x and yields the computable function $f = \{F_x\} : A \to B$, which can be applied to all data a of type A. To understand the step from (6) to (7), consider the category Pfn of partial computable

functions between the finite powers of the set of natural numbers $\mathbb{N}$, with $1 = \mathbb{N}^0$. Since $\mathsf{Pfn}(X, 1) = \wp X$, there is no terminal object, thus no cartesian structure. That is why cartesian products $\times$ from (6) are relaxed to the tensor products $\otimes$ in (7). To be able to copy and to delete the data, we must specify the data services explicitly. Note that this does not just recover the cartesian structure without the naturality requirement, as there are generally many nonstandard data services, already in the categories as standard as Rel [28]. While the partiality can be modeled within the cartesian structure [26], [9], going beyond the standard model of computation, and modeling the randomized and quantum computers, does seem to genuinely require nonstandard data services [10], [30], [31]. On the other hand, the program enumerations and transformations on the left hand side of (7) are required to be total and single valued; hence the restriction to $\mathcal{C}^\natural$ (denoted $\mathcal{C}_c$ in [10]). Anticipating that the computations will be typed, but that all programs as data will be of the same type, we also replace the exponent B^A in (6) by the type $\mathbb{P}$ of programs in (7). These intuitions motivate the formal definition of monoidal computer.

Definition 3.1: A *monoidal computer* is a strict symmetric monoidal category $\mathcal{C}$ with the following structure for all objects A and B in $\mathcal{C}$

(I) a data service $AA \overset{\varrho}{\rightleftarrows} A \overset{\top}{\to} I$

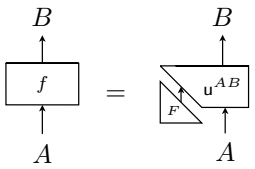
(II) a distinguished *object of programs* $\mathbb{P}$ and a family $\mathcal{C}^\natural(X, \mathbb{P}) \overset{\gamma_X^{AB}}{\twoheadrightarrow} \mathcal{C}(XA, B)$ of surjections natural in X. (δ)

3.2. Universal and partial evaluators

The families of surjections (7) turn out to be just a categorical version of the *acceptable enumerations* of computable functions [35], where the enumeration indices represent the programs.

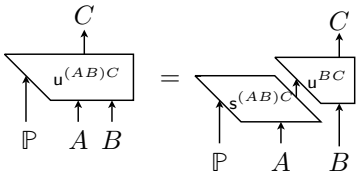
Proposition 3.2: Let $\mathcal{C}$ be a symmetric monoidal category with data services. Then specifying the surjections $\gamma_X^{AB} : \mathcal{C}^\natural(X, \mathbb{P}) \twoheadrightarrow \mathcal{C}(XA, B)$ that make $\mathcal{C}$ into a monoidal computer is equivalent to giving for all objects A and B in $\mathcal{C}$

(a) *universal evaluators* $\mathsf{u}^{AB} \in \mathcal{C}(\mathbb{P}A, B)$ such that for every $f \in \mathcal{C}(A, B)$ there is $F \in \mathcal{C}^\natural(\mathbb{P})$ (which we call a *program* for f) such that



(8)

(b) *partial evaluators* $\mathsf{s}^{(AB)C} \in \mathcal{C}^\natural(\mathbb{P}A, \mathbb{P})$ such that



(9)

Proposition 3.3: Every type A in a monoidal computer is a retract of $\mathbb{P}$: there are computations $\iota^A \in \mathcal{C}(A, \mathbb{P})$ and $\upsilon^A \in \mathcal{C}(\mathbb{P}, A)$, such that the composite $A \overset{\iota^A}{\longrightarrow} \mathbb{P} \overset{\upsilon^A}{\longrightarrow\!\!\!\!\!\rightarrow} A$ is the identity. These retractions are isomorphisms if and only if the family of surjections is natural in A or in B. Monoidal computer then provides a model of untyped λ-calculus.

Remark. In [32] we only considered the *basic* monoidal computer, where all types were the powers of $\mathbb{P}$. In the standard model, the programs are encoded as natural numbers, and all data are the tuples of natural numbers. Prop. 3.3 implies that all types must also be recursively enumerable in the internal sense of $\mathcal{C}$. In particular, by extending the λ-calculus constructions used in [32], we can extract from $\mathbb{P}$ the convenient types of natural numbers, truth values, etc. Here we only need the booleans. Deriving the boolean operations from the structure of the monoidal computer is an instructive exercise.

Definition 3.4: A *predicate* in a monoidal computer $\mathcal{C}$ is a computation $\alpha \in \mathcal{C}(A, 2)$, where the type 2 of *booleans* is a type where $\mathcal{C}^\natural(2) \cong \{0, 1\}$.

3.3. Examples of monoidal computer

The standard model of monoidal computer is the category of partial computable functions over the tuples of natural numbers. The programs are also encoded as natural numbers, and thus $\mathbb{P} = \mathbb{N}$. The objects can be just the powers of $\mathbb{N}$, but also all of their recursively enumerable subsets. The universal evaluators can be the computations implemented by a fixed family of universal Turing Machines. The partial evaluators are the total recursive functions constructed in Kleene's *s-m-n*-Lemma [19]. Extending this model to recursive relations and nondeterministic Turing machines leads to minor changes of the evaluation structure, but introduces many nonstandard data services [28], which can be used to encode nonstandard algorithms [31]. A *quantum* monoidal computer can be defined within the category of complex Hilbert spaces, with all linear maps as morphisms. The data services are provided by Frobenius algebras [11]. The category of functions with respect to these data services is equivalent with the category of sets and functions [10],

so the encoding of programs will be classical, and the same as in the standard monoidal computer. The universal and the partial evaluators can be defined as in [4]. It is important to note, however, that the program evaluations γ^{AB} are not surjective in a set-theoretic sense, but dense for the topological sense spelled out in [4]. Lastly, let us mention that any *reflexive domain* [13] gives rise to an *extensional* monoidal computer, which embodies both (6) and (7). For more detail see [32, Sec. 4.1].

4. Graded monoidal computer

4.1. Graded categories

While categories are very convenient for denotational semantics, capturing computation as a process requires some additional structure [1], [33], [22]. Capturing the complexity of computations requires a quantifiable view of that process, to allow us to count the steps, measure the memory, etc. If the morphisms of a monoidal computer represent computations, then we need to introduce some structure over these morphisms to express how much of a computational resource each of them uses. One idea is to consider the subsets $\mathcal{C}_n(A,B) \subseteq \mathcal{C}(A,B)$ that consist of those computations that use at most n units of a given resource: e.g., at most n steps in time, or n cells of space. Since the composite programs $p;q$ and $p \otimes q$ may need up to $m \oplus n$ units of the resource, if p needs m units, and q needs n, then the scale in which the resource will be measured must carry at least the structure of an additive monoid.

Grading monoids. Let $(\mathcal{M}, \oplus, 0, \infty)$ be a commutative monoid with the absorptive element ∞, i.e. such that $\infty \oplus m = \infty$ for all $m \in \mathcal{M}$. The relation

$$m \leq n \quad \iff \quad \exists \ell.\ \ell \oplus m = n \tag{10}$$

is obviously transitive and reflexive, i.e. a *preorder*. The equivalence classes with respect to the relation $m \sim n \iff m \leq n \wedge m \geq n$ are also the factors of the subgroup $\mathcal{G} = \{m \in \mathcal{M} \mid \exists n.\ m \oplus n = 0\}$. For simplicity, we assume $\mathcal{G} = \{0\}$, i.e. begin by factoring $\mathcal{M}$ modulo $\sim$. This not only makes $\leq$ into a partial order, but when $\mathcal{M}$ is finitely generated, then $(\mathcal{M}, \leq)$ is also a well founded lattice. For more see [36].

Examples of grading monoids. The additive monoid of natural numbers completed at the top $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ is the free grading monoid over one generator. On the other hand, the monoid of multisets of well-formed expressions in any given language, extended by ∞ again, can also be used as a grading monoid. In-between these extremes are the grading monoids normally used in complexity theory: the quotients $\overline{\mathbb{N}}^{\mathbb{N}} / \overset{+}{\equiv}$ and $\overline{\mathbb{N}}^{\mathbb{N}} / \overset{\mathcal{O}}{\equiv}$ of the monoid $\overline{\mathbb{N}}^{\mathbb{N}}$ of functions from $\mathbb{N}$ to $\overline{\mathbb{N}}$, identified modulo the equivalence relations

$$\begin{aligned} f \overset{+}{\equiv} g \quad &\iff \quad f \overset{+}{\leq} g \wedge f \overset{+}{\geq} g \\ f \overset{\mathcal{O}}{\equiv} g \quad &\iff \quad f \overset{\mathcal{O}}{\leq} g \wedge f \overset{\mathcal{O}}{\geq} g \end{aligned}$$

where

$$\begin{aligned} f \overset{+}{\leq} g \quad &\iff \quad \exists c\ \forall x.\ f(x) \leq c + g(x) \\ f \overset{\mathcal{O}}{\leq} g \quad &\iff \quad \exists cd\ \forall x \geq d.\ f(x) \leq c g(x) \end{aligned}$$

Definition 4.1: A category $\mathcal{C}$ is *$\mathcal{M}$-graded* if every hom-set $\mathcal{C}(A,B)$ comes with

(i) *grading* $\mathcal{C}(A,B) \xrightarrow{\|-\|} \mathcal{M}$, which induces the graded hom-sets

$$\mathcal{C}_n(A,B) \ = \ \{f \in \mathcal{C}(A,B) \mid \|f\| \leq n\} \tag{11}$$

the elements of which we write as f_n;

(ii) *restriction* $\restriction_n : \ \mathcal{C}(A,B) \to \mathcal{C}_n(A,B)$ for every $n \in \overline{\mathbb{N}}$ where for all $f_m \in \mathcal{C}_m(A,B)$ holds

$$(f_m)\restriction_n \ = \ f_{m \wedge n} \tag{12}$$

We require that identity morphisms are of grade 0, and

$$\|f\,;g\| \quad \leq \quad \|f\| \oplus \|g\| \tag{13}$$

which makes the following diagram commute

$$\begin{array}{ccc} \mathcal{C}_\ell(A,B) \times \mathcal{C}_n(B,C) & \xrightarrow{(\,;\,)} & \mathcal{C}_{\ell \oplus n}(A,C) \\ {\scriptstyle \mathrm{id} \times \restriction_m}\downarrow & & \downarrow{\scriptstyle \restriction_{\ell \oplus m}} \\ \mathcal{C}_\ell(A,B) \times \mathcal{C}_m(B,C) & \xrightarrow{(\,;\,)} & \mathcal{C}_{\ell \oplus m}(A,C) \end{array} \tag{14}$$

An *$\mathcal{M}$-graded monoidal category* is a monoidal category $\mathcal{C}$, which is $\mathcal{M}$-graded in the above sense, with all monoidal isomorphisms (such as $A \otimes B \xrightarrow{S} B \otimes A$) of grade 0, and moreover

$$\|f \otimes t\| \quad \leq \quad \|f\| \oplus \|t\| \tag{15}$$

making the following diagram commute

$$\begin{array}{ccc} \mathcal{C}_\ell(A,B) \times \mathcal{C}_n(P,Q) & \xrightarrow{\otimes} & \mathcal{C}_{\ell \oplus n}(AP, BQ) \\ {\scriptstyle \restriction_k \times \restriction_m}\downarrow & & \downarrow{\scriptstyle \restriction_{k \oplus m}} \\ \mathcal{C}_k(A,B) \times \mathcal{C}_m(P,Q) & \xrightarrow{\otimes} & \mathcal{C}_{k \oplus m}(AP, BQ) \end{array} \tag{16}$$

Remark. Note that we do not impose any requirements on the grading map $\mathcal{C}(A,B) \xrightarrow{\|-\|} \mathcal{M}$ that would allow lifting the lattice structure from $\mathcal{M}$ to $\mathcal{C}(A,B)$, and in general $\bigvee\{n < \infty\} = \infty$ does

not imply that $\bigcup_{n<\infty} \mathcal{C}_n(A,B)$ covers $\mathcal{C}_\infty(A,B) = \mathcal{C}(A,B)$. It usually does not.

Definition 4.2: We say that a computation $I \xrightarrow{a} A$ *halts* if there is $m < \infty$ such that $a = a\restriction_m$. More generally, we say that $A \xrightarrow{f} B$ *halts* if the composite $I \xrightarrow{a} A \xrightarrow{f} B$ halts whenever a halts. The subcategory of *total functions that halt* is denoted $\mathcal{C}_\natural$, and thus

$$\mathcal{C}_\natural(A,B) \quad = \quad \bigcup_{m<\infty} \mathcal{C}_m^\natural(A,B)$$

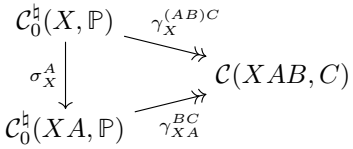
Definition 4.3: We say that a predicate $A \xrightarrow{\alpha} 2$ is *decidable* if it always halts and moreover outputs a single value, i.e. $\alpha \in \mathcal{C}_\natural$.

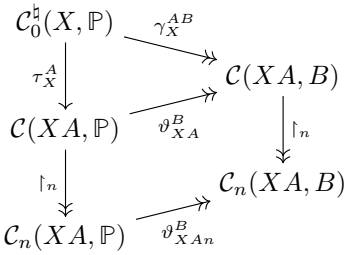
4.2. Definition and characterization

Definition 4.4: An $\mathcal{M}$-graded monoidal category $\mathcal{C}$ with data services in $\mathcal{C}_0^\natural$ is an *$\mathcal{M}$-graded monoidal computer* if it carries the following structure for all $A, B \in \mathcal{C}$:

(I) a family $\mathcal{C}_0^\natural(X,\mathbb{P}) \xrightarrow{\gamma_X^{AB}} \mathcal{C}(XA,B)$ of surjections natural in X,

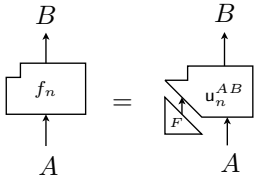
(II) families σ_X^A, τ_X^A and ϑ_X^A, all natural in X, such that the following diagrams commute





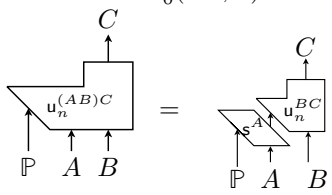
Proposition 4.5: Let $\mathcal{C}$ be an $\mathcal{M}$-graded monoidal category with data services. Then specifying the structure of an $\mathcal{M}$-graded monoidal computer as in Def. 4.4(I-II) is equivalent to giving for all $A, B \in \mathcal{C}$ and all $n \in \mathcal{M}$

(a) *graded universal evaluators* $\mathsf{u}_n^{AB} \in \mathcal{C}_n(\mathbb{P}A, B)$ such that for every $f_n \in \mathcal{C}_n(A,B)$ there is $F \in \mathcal{C}_0^\natural(X,\mathbb{P})$ such that



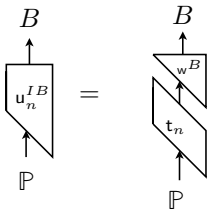
(17)

(b) *partial evaluators* $\mathsf{s}^A \in \mathcal{C}_0^\natural(\mathbb{P}A, \mathbb{P})$ such that



(18)

(c) *trace evaluators* $\mathsf{t}_n \in \mathcal{C}_n(\mathbb{P},\mathbb{P})$ and *output extractors* $\mathsf{w}^B \in \mathcal{C}_0^\natural(\mathbb{P}, B)$ such that



(19)

4.3. Examples of graded monoidal computer

First the bad news. Since graded monoidal computer extends the structure of monoidal computer, it seems natural that the standard model of monoidal computer, with a suitable grading, should provide a model of graded monoidal computer. However, the standard monoidal computer cannot be extended to a graded monoidal computer! The reason lies in Blum's Speedup Theorem [6], [7]. Blum's constructed a computable function such that for every program that implements it there is another program that computes the same, but faster by an arbitrary recursive factor. The construction applies to an arbitrary Blum measure (cf. Def. 6.1). The message is that complexity is not a property of computable functions, but of programs. Therefore, the category of computable functions, which provides the standard model of monoidal computer, is not a good place to measure complexity.

The good news is that this is not a bug, but an important feature of the approach! The concept of graded monoidal computer, as a categorical axiomatization of computational complexity, uncovers the fact that the universe of computation should not be modeled as the category of computable functions, but the category of computations, viewed as the pairs $\langle$program, function that it implements$\rangle$. The realisation that this is the right categorical model echoes what Levin and Meyer called the *"Fundamental Theorem of Complexity Theory"*[23], [25], which they formulated as a statement unifying Blum's Speedup and Compression Theorems [6], [7]. The standard model of graded monoidal computer thus presents a computation as a pairs $A \xrightarrow{\langle f,F\rangle} B$, where[1] $\{F\} = f$.

1. Recall that Kleene's bracket $\{-\}$ is used in classical resursion theory to denote the universal evaluators.

This is a monoidal computer, since for every computation $A \xrightarrow{\langle f,F\rangle} B$, i.e. for every program F, there is a program Φ such that $\{\Phi\} = F$ and a morphism $1 \xrightarrow{\langle F,\Phi\rangle} \mathbb{P}$, so that we can define $\gamma_I^{AB}(F,\Phi) = \langle\{F\},\{\Phi\}\rangle = \langle f, F\rangle$.

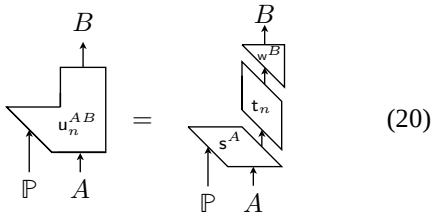
To define the grading, consider the set $\mathcal{T}$ of the execution traces like (1), and fix an encoding $\mathcal{T} \xrightarrow{\ulcorner - \urcorner} \mathbb{N}$ such that the parallel and the sequential compositions of the traces are associative and unitary. These requirements mean that the program compositions, the data-passing and buffering operations (both modeled by the identity morphisms), and the input-output operations are all assigned grade 0. Define a total recursive function $\mathbb{N}\times\mathbb{N} \xrightarrow{\oplus} \mathbb{N}$ so that $\ulcorner f\urcorner \oplus \ulcorner g\urcorner \geq \ulcorner f\,;g\urcorner$ and $\ulcorner f\urcorner \oplus \ulcorner t\urcorner \geq \ulcorner f \otimes t\urcorner$. Fix the universal evaluators u, and define $\mathsf{u}_n^{AB}(F,a)$ to not just build the trace $\mathsf{t}(F,a)$, but at the same time computes the code $\ulcorner \mathsf{t}(F,a)\urcorner \in \mathbb{N}$; and that it halts when $\ulcorner \mathsf{t}(F,a)\urcorner > n$. Let $\mathcal{C}_n(A,B)$ consist of the computations $\langle f, F\rangle$ that "clock out" at n, i.e. $\mathsf{u}(F,a) = \mathsf{u}_n(F,a)$ for all a.

While the details of the trace encodings and their use in the evaluations have to be deferred for the full paper, it should be clear that they amount to a routinely, albeit lengthy programming task, that awaits, e.g., the designer of a debugging tool that needs to capture, store, and play the internal execution traces.

The nondeterministic and the quantum monoidal computers lift in a similar way, but lead to substantially different grading structures.

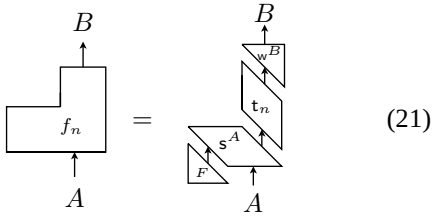
5. Normal form

Lemma 5.1: Every graded universal evaluator u_n^{AB} in a graded monoidal computer decomposes into the normal form $\mathsf{s}^A\,;\mathsf{t}_n\,;\mathsf{w}^B$, or diagrammatically:



(20)

Corollary 5.2: Every computation $f_n : A \to B$ in a graded monoidal computer decomposes for each of its programs F into the normal form $F\,;\mathsf{s}^A\,;\mathsf{t}_n\,;\mathsf{w}^B$



(21)

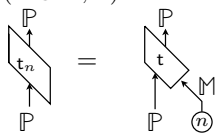
Remark For $n = \infty$, Corollary 5.2 is the monoidal version of Kleene's Normal Form Theorem [19, or any textbook in recursion theory]. Kleene proved his theorem by specifying a primitive recursive output extractor w^B, and implementing $\mathsf{t}_\infty\left(\mathsf{s}^A(F,a)\right)$ in the form $\mu x.T(F,a,x)$, where T is a primitive recursive predicate that verifies that x is the trace of the evaluation $\{F\}(a)$ of the program F on the input a. Corollary 5.2 just spells out that the categorical axioms of graded monoidal computer suffice for Kleene's decomposition. The point is that this decomposition, in a sense, displays the process of computation, encoded in the execution traces, as the grading of the trace evaluators t_n. This allows us to measure complexity.

6. Complexity measures

6.1. Internal grading

Intuitively, a complexity measure is a computable function c that takes a program F and an input a, and measures how much of a computational resource is needed to evaluate F on a. Since the measurements will be derived from the grades, we now assume that the grading monoid $\mathcal{M}$ is representable in $\mathcal{C}$. This means that there is an object $\mathbb{M}$ in $\mathcal{C}$ such that each $m \in \mathcal{M}$ is just a basis point $I \xrightarrow{m} \mathbb{M}$, i.e. $\mathcal{M} = \mathcal{C}_0^\natural(\mathbb{M})$. The monoid structure of $\mathcal{M}$ is internalized as the diagram $\mathbb{M}\otimes\mathbb{M} \xrightarrow{\oplus} \mathbb{M} \overset{0}{\leftleftarrows} I$ in $\mathcal{C}_0^\natural$. It is easy to see that the ordering $\leq$ of $\overset{\infty}{\mathcal{M}}$ is then also representable as an internal predicate $\mathbb{M}\otimes\mathbb{M} \xrightarrow{\leqslant} 2$ in $\mathcal{C}_0^\natural$.

We further assume that the grading of the trace evaluators $\mathsf{t}_n \in \mathcal{C}_n(\mathbb{P},\mathbb{P})$ is internalized in the sense that there is $\mathsf{t} \in \mathcal{C}(\mathbb{P}\otimes\mathbb{M},\mathbb{P})$ such that



(22)

The normalization now implies that the grading of the universal evaluators is similarly internalized. Note, however, that this does not imply that every sequence of computable functions $\langle f_n\rangle \in \prod_{n<\infty}\mathcal{C}_n(A,B)$ comes from some $f \in \mathcal{C}(A\otimes\mathbb{M},B)$. This would imply that a computable limit $f_\infty \in \mathcal{C}(A,B)$ with $(f_\infty)\restriction_n = f_n$ always exists, which is not the case, as many uncomputable functions have computable approximations [14]. Kolmogorov complexity provides the basic examples.

6.2. From Blum measures to normal measures

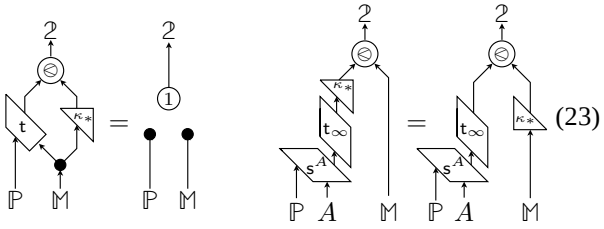
Definition 6.1: A *Blum measure* (or a *step-counting function* [6]) is a computation $\mathsf{c}^A \in \mathcal{C}(\mathbb{P}A,\mathbb{M})$ where

(i) $I \xrightarrow{F\otimes a} \mathbb{P}\otimes A \xrightarrow{\mathsf{c}^A} \mathbb{M}$ halts if and only if
$I \xrightarrow{F\otimes a} \mathbb{P}\otimes A \xrightarrow{\mathsf{u}^{AB}} B$ halts

(ii) $\mathbb{P}\otimes A\otimes\mathbb{M} \xrightarrow{\mathsf{c}^A\otimes\mathbb{M}} \mathbb{M}\otimes\mathbb{M} \xrightarrow{\circledcirc} 2$ is decidable.

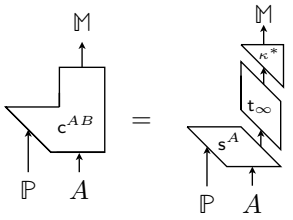
While condition (i) says that the abstract complexity measures are closely related with the universal evaluators, condition (ii) says that they are essentially different. The following definition attempts to capture that difference.

Definition 6.2: A *notion of complexity* is a triple $(\kappa^*, \kappa_*, \circledcirc)$ where $\kappa^* : \mathbb{P} \rightleftarrows \mathbb{M} : \kappa_*$ are halting functions[2] in $\mathcal{C}_\natural$, the predicate $\mathbb{P}\otimes\mathbb{P} \xrightarrow{\circledcirc} 2$ is in $\mathcal{C}_0^\natural$, and they together satisfy



(23)

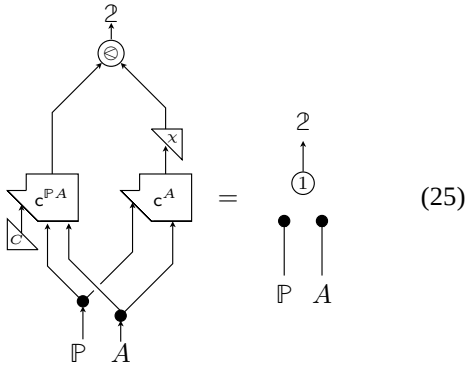
Definition 6.3: A *normal complexity measure* in a graded monoidal computer is a family of computations $\mathsf{c}^A \in \mathcal{C}(\mathbb{P}A, \mathbb{M})$ for all A in $\mathcal{C}$ for which there is a notion of complexity $(\kappa^*, \kappa_*, \circledcirc)$ that normalizes it:



(24)

Proposition 6.4: A normal complexity measure is a Blum measure where evaluating the complexity of a program is not substantially harder than evaluating the program itself. More precisely, there is $\chi \in \mathcal{C}_\natural(\mathbb{M}, \mathbb{M})$ and a program C for all c so that

2. The standard terminology in recursion theory is *"total recursive functions"*. But in other parts of mathematics, a function is always total on its domain, unless we don't know the domain, and it is specified that it is a partial. And moreover, a Turning machine may write some data on the output tape, and thus provide an output, without halting. So it seems necessary to specify that it is a halting function, and unnecessary to specify that it is total.



(25)

7. Summary

Theoretical computer science works with a wide gamut of different models of computation. While they all implement the same family of computable functions, they also confront us with a wide gamut of different low-level machine languages that we use to describe computations. The fact that all these different machines compute the same functions through radically different computational processes is usually celebrated as a conceptual miracle, giving rise to Church's Thesis. But the conceptual miracle gives way to the technical difficulties when it comes to programming in these machine languages. Proving that a feasible computation in one model will not become unfeasible in another model is generally not an easy task. With the questions that involve translating a complexity concept from one model to another often beyond reach, it seems fair to say that complexity theory is largely a mosaic of machine dependent concepts. The programming routines of *hiding the implementation details* and of *high-level languages*, on which the practice of computation has been based for more than 50 years, have not yet reached the theory of computation.

A notable early effort towards machine independent complexity theory was initiated by Blum [6], [7], and pursued by others [25], [23]. We implement and investigate some of their ideas in the framework of monoidal computer. Normal complexity measures, that admit a version of Kleene's normal form, emerge as an interesting concept. There is a sense in which the space of normal complexity measures is spanned by the time and the space complexity measures, as an orthogonal basis. This will have to be elaborated in the future work, together with all the proofs and details omitted from this extended abstract. While the measures that are not normal provide important design time information about algorithms, and for theoretical analyses, normal measures can also be used at run time, as practical tools of computation, e.g. to set the

bounds for hypothesis testing, inductive inference and algorithmic learning.

References

[1] S. Abramsky, "Retracing some paths in process algebra," in *CONCUR*, ser. Lecture Notes in Computer Science, U. Montanari and V. Sassone, Eds., vol. 1119. Springer, 1996, pp. 1–17.

[2] A. Asperti, "The intensional content of Rice's Theorem," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. New York, NY, USA: ACM, 2008, pp. 113–119.

[3] M. Bartha, "The monoidal structure of Turing machines," *Math. Struct. Comput. Sci.*, vol. 23, no. 2, pp. 204–246, 2013.

[4] E. Bernstein and U. V. Vazirani, "Quantum complexity theory," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1411–1473, 1997.

[5] L. Birkedal, "A general notion of realizability," *Bulletin of Symbolic Logic*, vol. 8, no. 2, pp. 266–282, 2002.

[6] M. Blum, "A machine-independent theory of the complexity of recursive functions," *J. ACM*, vol. 14, no. 2, pp. 322–336, Apr. 1967. [Online]. Available: http://doi.acm.org/10.1145/321386.321395

[7] ——, "On effective procedures for speeding up algorithms," in *STOC*, P. C. Fischer, S. Ginsburg, and M. A. Harrison, Eds. ACM, 1969, pp. 43–53.

[8] A. Carboni and R. F. Walters, "Cartesian bicategories, I," *J. of Pure and Applied Algebra*, vol. 49, pp. 11–32, 1987.

[9] J. R. B. Cockett and P. J. W. Hofstra, "Introduction to Turing categories," *Ann. Pure Appl. Logic*, vol. 156, no. 2-3, pp. 183–209, 2008.

[10] B. Coecke, Éric Oliver Paquette, and Dusko Pavlovic, "Classical and quantum structuralism," in *Semantical Techniques in Quantum Computation*, S. Gay and I. Mackie, Eds. Cambridge University Press, 2009, pp. 29–69.

[11] B. Coecke and D. Pavlovic, "Quantum measurements without sums," in *Mathematics of Quantum Computing and Technology*, G. Chen, L. Kauffman, and S. Lamonaco, Eds. Taylor and Francis, 2007, arxiv.org/quant-ph/0608035.

[12] S. Eilenberg and C. Elgot, *Recursiveness*, ser. ACM Monograph. Academic Press, 1970.

[13] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, and M. W. Mislove, *A compendium of continous lattices*. Cambridge University Press, 2003, (First edition 1980).

[14] E. M. Gold, "Limiting recursion," *J. Symb. Log.*, vol. 30, no. 1, pp. 28–48, 1965.

[15] P. J. W. Hofstra and M. A. Warren, "Combinatorial realizability models of type theory," *Ann. Pure Appl. Logic*, vol. 164, no. 10, pp. 957–988, 2013.

[16] J. Hyland, "The effective topos," in *The L.E.J. Brouwer Centenary Symposium*, A. Troelstra and D. V. Dalen, Eds. North Holland Publishing Company, 1982, pp. 165–216.

[17] A. Joyal and R. Street, "The geometry of tensor calculus I," *Adv. in Math.*, vol. 88, pp. 55–113, 1991.

[18] G. M. Kelly, *Basic concepts of enriched category theory*. Cambridge University Press, 1982.

[19] S. C. Kleene, "General recursive functions of natural numbers." *Math. Ann. 112, 727-742*, 1936.

[20] J. Lambek and P. J. Scott, *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1986.

[21] F. W. Lawvere, "Adjointness in foundations," *Dialectica*, vol. 23, pp. 281–296, 1969.

[22] M. Lenisa, J. Power, and H. Watanabe, "Category theory for operational semantics," *Theor. Comput. Sci.*, vol. 327, no. 1-2, pp. 135–154, 2004.

[23] L. A. Levin, "Computational complexity of functions," *Theoretical Computer Science*, vol. 157, no. 2, pp. 267 – 271, 1996.

[24] S. MacLane, *Categories for the Working Mathematician*, ser. Graduate Texts in Mathematics. Springer-Verlag, 1971, no. 5.

[25] A. R. Meyer and P. C. Fischer, "Computational speed-up by effective operators," *J. Symb. Log.*, vol. 37, no. 1, pp. 55–68, 1972.

[26] R. A. D. Paola and A. Heller, "Dominical Categories: Recursion Theory without Elements," *J. Symbolic Logic*, vol. 52, no. 3, pp. 594–635, 1987.

[27] D. Pavlovic, "Categorical logic of names and abstraction in action calculus," *Math. Structures in Comp. Sci.*, vol. 7, pp. 619–637, 1997.

[28] ——, "Quantum and classical structures in nondeterministic computation," in *Proceedings of Quantum Interaction 2009*, ser. Lecture Notes in Artificial Intelligence, P. Bruza, D. Sofge, and K. van Rijsbergen, Eds., vol. 5494. Springer Verlag, 2009, pp. 143–158, arxiv.org:0812.2266.

[29] ——, "Gaming security by obscurity," in *Proceedings of NSPW 2011*, C. Gates and C. Hearley, Eds. New York, NY, USA: ACM, 2011, pp. 125–140, arxiv:1109.5542.

[30] ——, "Relating toy models of quantum computation: comprehension, complementarity and dagger autonomous categories," *E. Notes in Theor. Comp. Sci.*, vol. 270, no. 2, pp. 121–139, 2011, arxiv.org:1006.1011.

[31] ——, "Geometry of abstraction in quantum computation," *Proceedings of Symposia in Applied Mathematics*, vol. 71, pp. 233–267, 2012, arxiv.org:1006.1010. [Online]. Available: http://www.comlab.ox.ac.uk//files/2533/RR-09-13.pdf

[32] ——, "Monoidal computer I: Basic computability by string diagrams," *Information and Computation*, vol. 226, pp. 94–116, 2013, arxiv:1208.5205.

[33] D. Pavlovic and S. Abramsky, "Specifying interaction categories," in *Category Theory and Computer Science '97*, ser. Lecture Notes in Computer Science, E. Moggi and G. Rosolini, Eds., vol. 1290. Springer Verlag, 1997, pp. 147–158.

[34] R. Penrose, "Structure of space-time," in *Batelle Rencontres, 1967*, C. DeWitt and J. Wheeler, Eds. Benjamin, 1968.

[35] H. Rogers, Jr., *Theory of recursive functions and effective computability*. Cambridge, MA, USA: MIT Press, 1987.

[36] M. P. Schützenberger, "On finite monoids having only trivial subgroups," *Information and Control*, vol. 8, no. 2, pp. 190–194, 1965.

[37] D. S. Scott, "Domains for denotational semantics," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, M. Nielsen and E. M. Schmidt, Eds. Springer, 1982, vol. 140, pp. 577–610.

[38] P. Selinger, "A survey of graphical languages for monoidal categories," in *New structures for physics*. Springer, 2011, pp. 289–355.

[39] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA, USA: MIT Press, 1977.

[40] S. Thompson, *Haskell - the craft of functional programming*, ser. International computer science series. Addison-Wesley, 1996.

Appendix

Proof sketch for Prop. 3.2

Given γ as in Def. 3.1(II), the universal evaluators u are defined

$$\mathsf{u}^{AB} = \gamma^{AB}_{\mathbb{P}}(\mathrm{id}_{\mathbb{P}}) \tag{26}$$

and then the partial evaluators s are determined by

$$\gamma^{BC}_{\mathbb{P}\otimes A}\left(\mathsf{s}^{(AB)C}\right) \;=\; \mathsf{u}^{(AB)C} \tag{27}$$

The other way around, if the universal evaluators u are given as in Prop. 3.2(a), then the natural transformation γ^{AB} as in Def. 3.1(II) can be defined by

$$\gamma^{AB}_X\left(X \xrightarrow{h} \mathbb{P}\right) \;=\; XA \xrightarrow{hA} \mathbb{P}A \xrightarrow{\mathsf{u}^{AB}} B \tag{28}$$

This is surjective because

- 3.2(a) says that every $g \in \mathcal{C}(XA, B)$ decomposes to

  $$XA = IXA \xrightarrow{\widetilde{g}XA} \mathbb{P}XA \xrightarrow{\mathsf{u}^{(XA)B}} B$$

  for some $\widetilde{g} \in \mathcal{C}^{\natural}(I, \mathbb{P})$, whereas
- 3.2(b) and (28) give

  $$\gamma^{AB}_X\left(IX \xrightarrow{\widetilde{g}X} \mathbb{P}X \xrightarrow{\mathsf{s}^{(XA)B}} \mathbb{P}\right) \;=\; g$$

Proof sketch for Prop. 4.5

Given the natural transformations as in Def. 4.4, the evaluators from Prop 4.5 are defined by

$$\mathsf{u}^{AB}_n \;=\; \gamma^{AB}_{\mathbb{P}}(\mathrm{id}_{\mathbb{P}})\restriction_n \tag{29}$$

$$\mathsf{s}^{A} \;=\; \sigma^{A}_{\mathbb{P}}(\mathrm{id}_{\mathbb{P}}) \tag{30}$$

$$\mathsf{t}_n \;=\; \tau^{I}_{\mathbb{P}}(\mathrm{id}_{\mathbb{P}})\restriction_n \tag{31}$$

$$\mathsf{w}^{B} \;=\; \vartheta^{B}_{\mathbb{P}}(\mathrm{id}_{\mathbb{P}}) \tag{32}$$

It is not hard to check that the conditions of Def. 4.4 are equivalent with the conditions of Prop. 4.5.